**小程序**

**小程序页面级别的生命周期：**

onload():页面创建完 只触发一次 通常获取后台数据或接收另一个页面的传参时使用

onShow():进入页面触发

onHide():页面离开时触发

onReady():页面初次渲染完成

onUnload():监听页面卸载，类似于vue中的destory

onPullDownRefresh():下拉触发

onShareAppMessage():点击右上角分享时触发

**购物车业务逻辑**

通过观察可以发现，淘宝的购物车商品及其数量在不同设备中同步更新，

但是购物车商品的勾选和核算状态因设备而异。

我实现的电商小程序中购物车有如下功能：

1. 商品详情页中点击加入购物车，
2. 增加或减少购物车中某商品的件数
3. 勾选/全选/取消勾选商品，同时核算的价格发生变化

通过设置一个全局变量cart，作为缓存的购物车数组。

请求过来的购物车数组中，每个商品对象没有表示checked勾选的字段，

以最新请求过来的购物车数组为标准，通过find比较id来查找缓存cart中是否有该商品。如果缓存cart中没有该商品，就往缓存添加该商品；如果找到缓存cart中有该商品obj，则把缓存cart的数量num修改为最新请求过来的num，再push到新的缓存购物车数组copycart

**在onLoad()生命周期中调用接口，获取该用户的购物车数据**

**因为增加或减少商品，在不同设备中能同步更新。所以我会选择发起请求，把要添加或减少的商品id传给后端，响应回来的购物车数据重置缓存**

**CSS**

**position属性**

1. static:position的默认值，元素处于正常的文档流中。Top、right、bottom、left等属性失效
2. relative对象遵循标准文档流中，依赖top、right、bottom、left等属性相对于该对象在标准文档流中的位置进行偏移，同时可以通过z-index定义层叠关系
3. absolute:对象脱离标准文档流，使用top、right、bottom、left等属性相对于static定位以外的第一个父元素进行绝对定位，同时可通过z-index定义层叠关系
4. fixed：对象脱离标准文档流，使用top、right、bottom、left等属性进行绝对定位，同时可以通过z-index定义层叠关系

**CSS权重**

从0开始，一个行内样式+1000，一个id选择器+100，一个属性选择器、class或者伪类+10，一个元素选择器或者伪元素+1，通配符+0

!import>行内样式>id选择器>属性选择器/class选择器/伪类>元素选择器/伪类>元素选择器/伪元素>通配符

**Flex布局**

Flex属性是flex-grow、flex-shrink、flex-basis的简写，默认值是0、1、auto

Flex-grow：定义项目的放大比例，默认0

Flex-shrink：定义项目的缩小比例，默认1，即如果空间不足，该项目将缩小

Flex-basis：给上面两个属性分配多余空间之前，计算项目是否有多余空间之前，计算项目是否有多余空间，默认值为auto，即本身的大小

Flex：1完整写法：flex-grow:1;flex-shrink:1;flex-basis:0%

**行内元素和块元素**

块元素：自动换行，横向充满其父元素的内容区域，默认独占一行，可修改宽高

Html、body、div、header、footer、nav、section、aside、article、p、

ul、ol、form、table

行内元素：不自动换行，行内元素左右可以有其它元素，行内元素的宽高大多是auto\*auto

Span、a、img、textarea、button、input、em、strong、label

**盒模型的区别**

border-box：这里的内容宽度/高度包含了元素的border、padding、内容的宽度或高度

content-box：元素的宽度和高度等于元素

**Vue**

**Vue.set方法原理**

**Vue2响应式原理的缺陷有两点**

1.在实例创建之后添加新的属性到实例上不会触发视图更新

原因：在Vue实例初始化的时候就对data全部属性递归observe,进行响应式处理。所以新添加的属性没有响应式处理。不会触发视图更新。

2.直接更改数组下标来修改数组的值不会触发试图更新

一是因为Object.defineProperty这个api无法监听数组长度的变化

不能监听数组长度变化的原因是数组的length的configurale为false

各大浏览器厂商和JS引擎规定不允许修改它。

二是因为数组长度可能很长，考虑到性能消耗和用户体验。

**Diff算法采用同级比较**

1. tag标签不一致直接将旧节点替换成新节点
2. tag标签一样

先替换属性

对比子元素

新旧节点都有子元素，采用双指针进行对比

sameVnode判断tag和key完全相同为同一节点，进行节点复用

头和头相等对比

尾和尾相等对比

头和尾相等对比

**sameVnode的时候传入两个新老子节点patch(oldChild,newChild)**

**$router和$route的区别**

Router是Vue Router的实例，相当于一个全局的路由器对象，里面有很多属性和子对象。

This.$router.push会往history栈中添加一个记录

Route是路由信息对象，可以从里面获取name、path、params、query等。

**Vue内置指令**

v-model：value和input的语法糖

v-for:key是Vue中Vnode的唯一标识，通过这个key，diff操作可以更准确更快速。

注意增加唯一key值，使用index作为key无法patch

v-bind:绑定属性 动态更新HTML元素上的属性

v-on：用于监听DOM事件

v-html:赋值就是变量的innerHTML 防止xss攻击

v-once:定义它的元素或组件只渲染一次，包括元素或组件的所有节点， 首次渲染后，不再随数据的变化重新渲染，将被视为静态内容

v-if/v-else-if/v-else

v-show

**Vue-router模式**

1. **hash模式**

hash模式的路径上‘#’

hash值不包含在请求中，所以改变hash值时不会刷新页面也不会发送新的请求

hash值的改变会触发hashchange事件（onhashchange 事件在当前 URL 的锚部分(以 '#' 号为开始) 发生改变时触发 。）

1. **history模式**

利用H5 History Interface中新增的pushState()和replaceState()方法

可以保存历史记录，方便前进和后退

依赖于后台配置，前端的url必须和实际后端发起请求的url一致，如果后来缺少对该url的路由处理，将返回404错误状态码

**路由独享守卫：写在路由对象中，只在这个路由下起作用**

**Vue路由守卫触发顺序**

1. 导航被触发(A->B)
2. 调用A组件内路由守卫beforeRouteLeave
3. 调用全局路由前置守卫beforeEach （一般用于登录前验证身份）
4. 在重用的组件里调用beforeRouteUpdate（从/foo/1和/foo/2之间跳转需要重新请求数据，这种类型不会触发created生命周期函数，可以通过该钩子函数或监听$route来实现）
5. 调用B路由独享守卫beforeEnter
6. 解析异步路由组件B
7. 调用B的组件内路由守卫beforeRouteEnter
8. 调用全局路由解析守卫route.beforeResolve
9. 导航被确认
10. 调用全局路由钩子router.afterEach
11. 渲染B组件DOM

**路由跳转的方式（编程式导航、router-link to、 router-link :to）**

1. name和parmas，parmas拼接在url上
2. path和query
3. url跳转

**v-if和v-show的区别**

v-show和v-if都能控制元素的显示隐藏

v-show只编译一次，初始一定会渲染，后面通过display:none控制隐藏。初始渲染开销大，切换开销小

v-if是懒加载，初始为false时不会渲染，动态的向DOM树内添加或删除DOM元素，不停的销毁和创建。初始渲染开销小，切换开销大

要频繁切换某个节点用v-show，

**v-model和v-bind的区别**

**v-model：**实现了双向绑定。是v-bind和v-on的结合。

给组件添加v-model属性默认把value作为组件的属性，然后把input值作为组件绑定事件时的事件名。双向绑定是在单向绑定的基础上给可输入元素添加change事件，来动态修改model和view，即通过触发$emit父组件的事件来修改mv来达到mvvm的效果。而vue组件间传递数据是单向的，数据从父组件传到子组件，子组件不能修改父组件传递的数据。这样做是为了组件间更好的解耦，假如子组件可以修改父组件的数据，那么所有依赖于这个父组件数据的子组件也会发生变化。

**v-bind:**实现了数据单向绑定，视图模型发生变化，视图也会发生变化。但是视图的变化并不能影响视图模型。

**当在子组件去修改props时**，可以这么做

1. 定义一个局部变量，并用prop的值初始化。
2. 定义一个计算属性，处理prop的值并返回。

**子组件可以改变父组件的数据吗？**

不可以。Vue提倡单向数据流，父级props的更新会流向子组件，但是反过来则不行。这是为了防止意外的改变父组件状态，使得应用的数据流变得难以理解，导致数据流混乱。只能通过$emit派发一个自定义事件，父组件接收到后，由父组件修改。

**Vuex的原理和自己的理解**

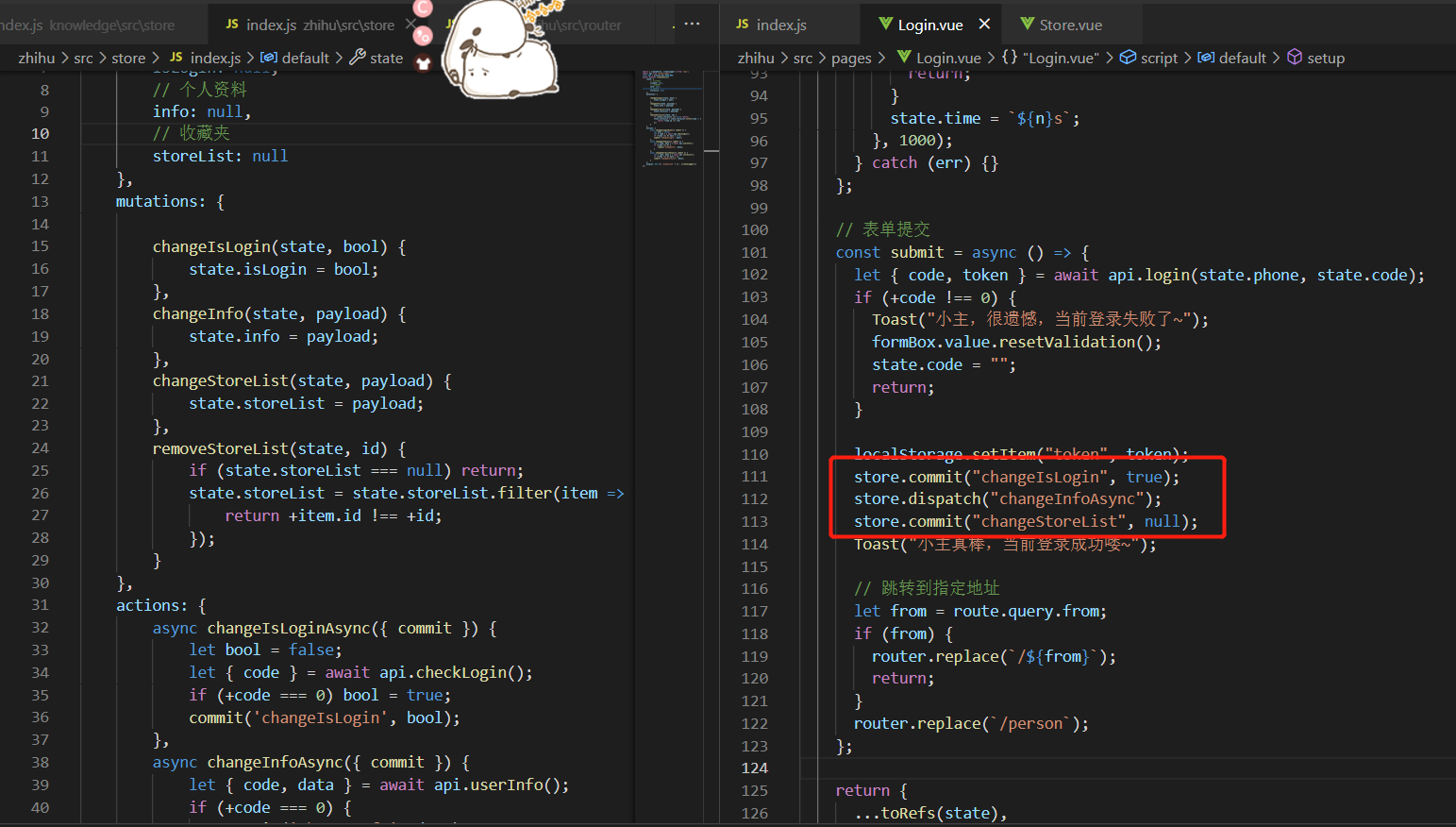
Vuex是一个专为Vue.js应用程序开发的状态管理模式。它采用集中式存储应用的所有组件的状态，并以相应的规则保证状态以一种可预测的方式发生变化。

**核心流程**：

1. Vue组件触发(dispatch)一些事件或动作，也就是actions
2. 在组件中发出动作，不能直接更改数据，要把这个动作提交到Mutations中
3. 然后Mutations去改变state中的状态
4. 当state中的状态被改变时，会重新渲染到Vue组件中去，组件展示更新后的数据，完成一个流程

在本人项目中对Vuex的使用：

我分别对登录状态、个人资料、收藏夹进行了状态管理



在【知乎日报APP】项目中创建了一个Vuex.Store对象来保存应用启动时的初始状态

有isLogin、StoreList和Info分别表示是否登录、收藏夹列表和个人信息

**【登录页】登录成功**之后

1.通过commit 一个changeIsLogin Mutation更改isLogin状态

2.通过dispatch触发changeInfoAsync异步方法，changeInfoAsync里异步请求用户个人信息，把响应结果通过commit修改vuex.store中的个人信息Info状态。

3.通过commit一个 changeStoreList Mutation更改StoreList状态

【个人中心页】

1. 先通过store.state获取用户的登录状态和个人信息
2. 如果没有登录就dispatch触发changeIsLoginAsync检查是否登录并通过commit mutation修改登录状态
3. 如果个人信息为null就dispatch触发changeInfoAsync请求用户个人信息并通过commit mutation修改vuex.store中用户的个人信息

【退出登录】

通过commit mutation把所有状态置空。

Vue组件：负责接收用户操作等交互行为，执行dispatch方法触发对应的action

Dipatch:操作行为触发方法，是唯一能执行action的方法

Actions:操作行为处理模块。包含同步/异步操作，支持同名方法，按照注册的顺序依次触发。

Commit:状态改变提交操作方法，对mutation进行提交，是唯一执行mutation的方法

Mutations:状态改变操作方法，是Vuex修改状态的唯一推荐方法

State:页面状态管理容器对象，集中存储Vue组件中data对象的零散数据，全局唯一，以进行统一的状态管理。

Getter:state对象读取方法。

**Vue中的通信方式有哪些？**

**父子通信、自定义属性、props、$emit、EventBus、$on、VueX**

【参考答案】

标准回答

Vue组件的通信方式分为两大类，一类是父子组件通信，另一类是任何关系类型组件通信（父子、兄弟、非兄弟）

**父子组件**的通信方式：

• **父给子传递数据，通过给子组件添加自定义属性**，比如：<List :list="list"/>，list是父组件给子组件传递的数据。子获取父的数据，在子组件中使用props属性获取

• **子给父传递数据，子组件使用$emit发送一个自定义事件**，父组件使用v-on监听子组件发送的事件

• props是只读，不可以被修改，所有被修改都会失效和被警告

**$refs可以获取子组件实例（只读）**

Ref只有等页面加载完成后才能调用this.$refs。

可以加个定时器延时加载this.$refs或者使用this.$nextTick(() =>{ })

**任何关系类型组件通信（父子、兄弟、非兄弟）方式**：

**Provide/inject:**

这对选项需要一起使用，以允许一个祖先组件向其所有子孙后代注入一个依赖，不管组件层次多深，在父组件的生命周期内，子孙组件都可以调用。

**EventBus**：

使用方法是创建一个新的Vue实例，需要通信的组件都引入该Vue实例，传递数据的组件使用 event.$emit('名称',参数)发送数据，接收数据的组件使用 event.$on('名称',方法)接收数据。EventBus的优缺点：缺点vue是单页应用，如果你在某一个页面刷新了之后，与之相关的EventBus会被移除，这样就导致业务走不下去。同时如果页面中有反复操作的业务，EventBus在监听的时候就会触发很多次，需要好好处理EventBus在项目中的关系。在vue页面销毁时，同时移除EventBus事件监听。优点，解决了多层组件之间繁琐的事件传播，使用原理十分简单，代码量少。适合简单单，组件传递数据较少的项目，大型项目业务复杂的还是尽量使用VueX

**VueX**：

集中管理项目公共数据，Vuex 的状态存储是响应式的，当 Vue 组件从 store 中读取状态的时候，若 store 中的状态发生变化，那么相应的组件也会相应地得到高效更新。不能直接改变 store 中的状态。改变 store 中的状态的唯一途径就是显式地提交 (commit) mutation。

**localStorage、sessionStorage、indexDB、cookie**

localStorage：持久化的本地存储，**以文件的形式存在本地(硬盘)**，除非强制删除就永远不会过期，不提供搜索功能，不能建立自定义索引，只能存字符串类型的数据，存储对象需要JSON的stringify和parse方法进行处理。跨页面传递数据时用。

sessionStorage:是本地的一个会话级别存储，在页面打开的时候创建，在页面关闭的时候销毁。

Cookie：大小受限，只有**4kb**的大小。服务器端和浏览器。并且每次发送一个新的页面的时候cookie都会发送过去。无形浪费了带宽；cookie还需要指定作用域，不可以跨域调用。

indexDB属于异步操作，存储空间大，**一般不少于250MB**,会受到同源限制；不仅可以储存字符串还可以储存二进制数据

Vuex：**Vuex存储在内存**。读取内存比读取硬盘速度要快， Vuex能做到数据的响应式，localStorage不能。刷新页面时Vuex存储的值会更新。

**Vue生命周期钩子**：

beforeCreated：实例初始化之后，数据观测和event/watcher事件配置之前被调用。这个阶段data、methods、computed和watch上的数据和方法都不能被访问

created：实例已经创建完之后被调用。这一阶段已经完成了数据观测、属性和方法的运算

beforeMount:模板编译/挂载开始之前被调用，相关的render函数首次被调用

mounded:在挂载完成之后，真实的DOM挂载完毕，数据完成双向绑定，可以访问到DOM节点

beforeUpdated:数据更新时调用，发生在虚拟DOM重新渲染和patch之前

updated:组件更新之后

activated:keep-alive组件激活前

deactivated:keep-alive 组件移除时调用

destroy:组件销毁后调用

activated:keep-alive专属，组件被激活时调用

deactived:keep-alive专属，组件被销毁时调用

**Vue2对数组的检测怎么处理**

Object.defineProperty不会对数据的索引进行劫持。Vue2重写了数组的方法，通过Object.create()创建一个新的数组原型方法，不影响数组原有的原型属性。对于数组的处理。

**Vue中的data为什么必须是个函数？**

一个组件的data选项必须是个函数，组件实例化的时候这个函数会被调用，返回一个对象，计算机会给这个对象分配一个内存地址，实例化几次就分配几个内存地址，他们的地址都不一样，所以每个组件中的数据互不干扰。

**Vue前端项目性能优化**

1. data对象层级不过深，避免Vue组件初始化时深层递归该对象属性处理响应式消耗性能
2. 不需要响应式的数据不要放在data中（Object.freeze()冻结数据）
3. 需要频繁隐藏和显示的用v-show，否则用v-if
4. computed和watch区分使用场景
5. 图片懒加载
6. 路由懒加载
7. 防抖、节流。

使用Object.defineProperty不会对数据的索引进行劫持，因为数组是

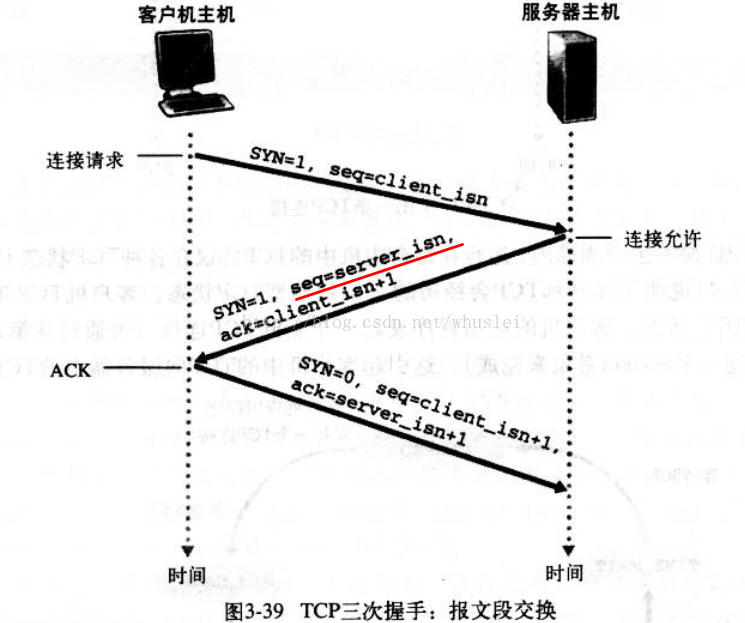
**Vue2和Vue3的区别**

1. Vue2的响应式原理：数据初始化initData时通过Observe类观测用户设置的data，通过递归操作将对象中所有的key重新defineProperty定义为响应式的，观测过的数据添加一个标识，说明已观测过，只对未观测过的检测。需要手动对新增对象进行观测。

**从输入url到浏览器显示页面发生了什么?**

当用户在浏览器的用户界面的地址栏上url，按下回车。会先对URL进行解析。其主要目的是通过查询将URL中的Host字段转化为网络中的IP地址，因为域名只是为了方便即的，IP地址才是所访问服务器在网络中的“门牌号”

依次查找浏览器自身的和系统自身的DNS，如果还未找到，就会尝试从系统的hosts文件中查找。在本地主机进行查询若没有获取到，接下来便会在本地域名服务器中查询，如果本地域名服务器没有直接的目标IP地址可供返回，则本地域名服务器会才去迭代的方式一依次查询根域名服务器、COM顶级域名服务器和权限域名服务器等等，最终将要访问的目标服务器的IP地址返回本地主机，若查询不到，则返回报错信息。外网IP地址是不会重复的。所以查询出IP地址，就可以找到资源所在的服务器，请求该服务器下指定的资源。客户端向服务器发送http请求，先通过三次握手，建立一个到服务器的TCP连接。客户端请求建立TCP连接，进行第一握手时客户端向服务器发送同步标志位SYN = 1，和一个随机的客户端初始序列号seq = j；服务器收到请求报文之后，向客户端发送SYN = 1，确认号ack = j + 1，还有一个seq = k，以表示收到了客户机的连接请求并允许客户机的连接请求。最后客户端收到服务器发来的请求确认之后，需要向服务器发送同步序号SYN = 0 ,seq = j + 1,ack =k + 1 。三次握手的目的在于确认客户端和服务器（收发）双方都有收发报文的能力。如果只有两次握手，客户端在某次发送了一个连接请求，但是该请求报文网络延迟了，久久收不到服务器的确认，于是客户端就会重新发送一个新的连接请求，结果网络畅通时服务器就收到了两个请求，建立起两个请求相同资源的请求连接，这样就会有一个连接浪费了服务器资源，达到一个效果多做了一次无用功。

  
 当TCP连接建立好之后，便可通过HTTP等协议进行前后端的通信，但在实际的网络访问中，发送请求的目标服务器一定是解析出的IP地址所在的服务器吗？并非如此，往往会在浏览器和确定IP地址的服务器之间加入反向代理。Web缓存器也叫代理服务器，它是能够代表Web服务器来满足HTTP请求的网络实体。 **Web缓存是什么？**Web缓存是可以自动保存常见文档副本的HTTP设备。当Web请求抵达缓存时，如果有本地缓存的副本，就有机会（还要判断是否可用）从本地缓存中提取这个文档。

**这样做有什么好处？**1.**冗余的数据传输**。很多客户端频繁访问同一个服务器页面时，服务器会多次传输同一个文档，相同的字节在网络中重复传输，这些冗余的数据传输会消耗网络带宽、降低传输速度、加重Web服务器的负载。保留缓存副本就可以减少这些损失。2.**带宽瓶颈。**很多网络为本地客户端提供的带宽远宽于提供给远程服务器的带宽。如果客户端从一个快速局域网的缓存中得到一个副本就可以提高性能了。3.**瞬间拥塞**。很多人同时向服务器访问一个Web文档就会给服务器造成巨大的压力，缓存减轻了服务器的压力。4.**距离时延。**即使客户端和浏览器之间没有太多的路由器，光速也会造成显著的时延。将缓存放在附近就会减少时延。

**缓存的处理步骤**：1.缓存接受http请求。2.缓存解析该报文，提取URL和各种首部。3.缓存查看本地是否有副本，如果没有就去请求源服务器获取一份副本并保存在本地。4.有了缓存副本还不行，要检查副本的有效性。HTTP1.0的产物expires字段是服务器响应头字段，它给出了一个副本过期的绝对时间。由于是绝对时间，由于请求需要时间，所以浏览器的请求时间与服务器接收到请求所获取的时间是存在误差的，从而导致浏览器判断缓存失效。已知Expires的缺点之后，HTTP1.1新增了Cache-control字段。常见的Cache-control的值有以下：

1. max-age：定义了文档的最大有效时间。以时间间隔标志失效时间，解决了Expires服务器和客户端相对时间的问题。
2. no-cache：不强缓存，要服务器再验证（协商缓存）
3. no-store：绝对禁止缓存对响应进行复制。
4. public：客户端和代理服务器都能缓存
5. private：只能客户端缓存

Cache-control的优先级高于Expires。

缓存再验证（协商缓存）中使用两个条件首部。

1. If-Modified-Since:<date> ：可以和服务器响应首部Last-Modified配合使用，原始服务器会将最后修改日期附加到所提供的文档，如果从指定日期之后文档被修改过，就要向服务器请求。
2. If-None-Match:<tags>：服务器可以为文档提供特殊的标签标志该文档，如果已缓存的标签和服务器文档标签不一样，就要向服务器请求。

**有些情况下使用If-Modified-Since进行再验证是不够的**。比如

* 更新间隙发生在一秒粒度单位以下。
* 有些文档被周期性重写，即使内容无变化
* 有些修改不重要。不需要让缓存替换副本。
* 有些服务器无法准确判断页面的最后修改日期

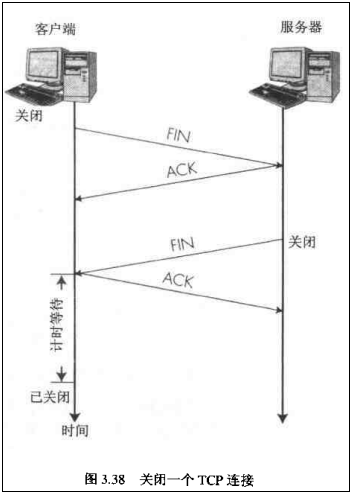
为了解决这些问题，HTTP允许用户对Etag进行比较。当发布者对文档进行修改时，可以修改文档的实体标签来说明这个新的版本，这样，如果实体标签被修改了，缓存就可以用If-None-Match条件首部来GET文档的新副本。

**强缓存和协商缓存分别适用于哪些场景**？强缓存优先于协商缓存进行，若强缓存生效则直接使用缓存，若不生效则进行协商缓存。协商缓存由服务器决定是否使用缓存，若协商缓存失效，那么代表该请求的缓存失效，返回200，重新返回资源和缓存标识，再存入浏览器缓存中；生效则返回304，继续使用缓存。

* 对于频繁变动的资源，首先需要使用Cache-Control：no-cache使浏览器每次都请求服务器，然后配合Etag或者Last-Modified来验证资源是否有效。这样的做法虽然不能节省请求数量，但能显著减少响应数据大小。
* 处理不常变化的资源时，给它们的Cache-Control配置一个很大的max-age。这样浏览器之后请求相同的URL会命中强制缓存。而为了解决更新的问题，就需要在文件名（或者路径）中添加hash，版本号等动态字符，之后更改动态字符，从而达到更改引用URL的目的，让之前的强制缓存失效。

服务器处理请求并返回HTTP报文。浏览器内核拿到内容后，1.**解析html**。通过字节流把内容传到html解析引擎，解析引擎先调用解析器处理标签，处理没闭合或使用不规范的标签，处理完毕之后交给分词器处理。分词器将这些符合规范的标签转换成token，并交给dom构建器处理。在解析过程中遇到script标签会暂停解析，阻塞解析html的进程。直到script加载完成并执行完，html才会继续解析。2.**构建dom树**。逐行解析代码，包括html标签和js动态生成的标签，将token解析成dom节点，并将dom节点添加到dom树中。通过压栈出栈的方式构建dom树。3.**构建css树**。在解析过程中遇到css资源需要加载时，html解析不会暂停。当css资源下载好，会调用css解析引擎解析CSS代码，计算出最终样式。对CSS代码中非法的语法会直接忽略。在解析css的同时，渲染树构建被暂停，直到css树解析完成。4.**渲染树。**把之前构建好的css树和dom树按照一定的规则和依赖关系生成渲染树，写到内存，等待进一步生成界面。渲染树和dom树是有区别的。Dom树完全和html标签一一对应，但是渲染树会忽略掉不需要渲染的元素。比如head、display:none的元素等。5.**布局**。从渲染树的根节点开始遍历，然后确定每个节点对象在页面上的确切大小和位置，布局阶段的输出是一个盒子模型，它会精确地捕获每个元素在屏幕内的确切位置与大小。同时把每个节点做成一个纹理，放在gpu寄存器中管理。6.**绘制**。遍历渲染树，调用渲染器的print()方法在屏幕上显示。

当数据传输完毕，需要断开TCP连接。建立TCP连接需要三次握手，断开TCP连接需要四次挥手。首先客户端发送FIN报文请求中断，“我要和你断连”，服务端收到FIN报文时服务端的数据也许还没有发送完毕，服务端先回复确认“收到，我知道了，先等我一下”，等服务端的数据发送完毕了，他才会发送一个FIN报文同意断连，“好了，拜拜”，客户端确认断连，“拜拜啦”。

****

每次发送请求都要重新建立TCP连接吗？不是的，这要看HTTP请求报文中请求行的HTTP版本。以下是我所了解到的各个**HTTP版本的特性**。

1. HTTP1.0
2. 在HTTP1.0中主要使用header里的**If-Modified-Since和Expires**作为缓存判断的标准。
3. **HTTP1.0不支持断点续传功能**，一次性只能发送整个对象。
4. **HTTP1.0**中认为每台服务器绑定一个唯一的IP地址，因此请求消息中的URL**没有传递Host Name**。但随着虚拟机的发展，在一台物理服务器上可以存在多台虚拟服务器，它们共享一个IP地址。
5. **HTTP1.0不支持长连接**，每次请求都要重新创建连接消耗资源。

二、HTTP1.1

1. **新增Cache-control、If-None-Match等可供选择的缓存头**来控制缓存策略
2. **在请求头引入range域，允许断点续传**。
3. **请求和响应头支持Host头域**。请求消息中如果没有Host头域会报错。
4. **支持长连接和请求流水线处理。HTTP1.1默认开启Keep-alive**，减少了建立和关闭的消耗和延迟。
5. **新增了24个错误状态响应码。如409表示请求的资源与资源的当前状态发生冲突；410表示服务器上的某个资源被永久性删除。**

HTTP1.x存在的问题：

1. **所有的传输都是明文传输**，客户端和服务端无法验证对方的身份，**造成数据不安全性。**
2. **HTTP1.x在使用时，header携带的内容太大**，增加传输成本，并且每次请求时**header基本不变化**，尤其在移动端会增加用户流量。
3. HTTP1.1虽然支持Keep-alive，弥补多次创建连接产生的时延。但是K**eep-alive可能会对性能产生极大的负面影响。比如它在文件被请求之后还保持了很长时间的不必要连接。**
4. 持久连接是默认开启的，虽然持久连接可以减少TCP的建立和断开次数，但是所有的请求和响应都是按序进行的。即便是HTTP1.1引入了管道机制，也是如此，客户端在未收到之前发出的所有请求之前，将会阻塞后面的请求。这就是**队头阻塞**
5. 。

三、HTTP2

1. **二进制传输**。HTTP1.x是基于文本的。HTTP2中所有传输数据都会被分割，并采用二进制格式编码。
2. **多路复用**。HTTP2基于二进制流，它可以把HTTP消息(stream)分解为独立的帧（frame），交错发送，然后在另一端通过帧中的标识(Sream Identifier)重新组装得到完整的请求。这就实现了在同一个TCP连接中，同一时刻可以发送多个请求和响应，且不用按照顺序一一对应，即使某个请求任务耗时严重，也不会影响到其它连接的正常进行。
3. **压缩header**。在HTTP1.x中，使用文本的形式传输header每次请求都会带上，这些基本上是不变的，如果再加上cookie的话也是不小的浪费。在HTTP2中，使用了HPACK压缩格式对header进行了编码，减少了header的大小。大概原理就是在服务端和客户端共同维护一个静态字典，用于记录出现过的header，后面在传输过程中发送端直接传输已经记录过的键名，接收端就可以通过键名找到对应的值。
4. **服务端push**。服务端可以在客户端的某个请求后，根据这个请求，主动推送其它资源。比如一个html页面中还带上一个css和js的资源请求，在http1.x时就需要发送三次资源请求。而http2中，服务器发现html中包含css和js资源，便会将这三个资源都返回给客户端，这样只需要通信一次，就可以获得全部资源。

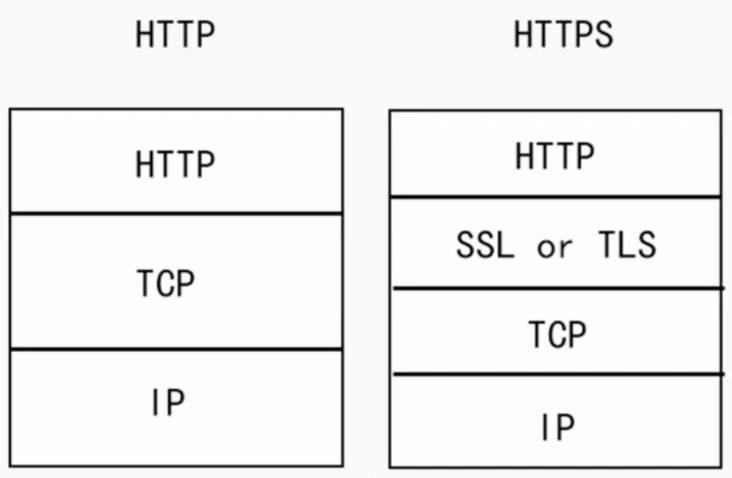
HTTP是超文本传输协议，是客户端浏览器或其他程序和Web服务器之间的应用层通信协议。HTTP请求过程数据明文传输，容易遭到黑客的攻击。HTTP传输面临的风险：

1. 窃听风险：黑客可以获取通信内容。
2. 篡改风险：黑客可以修改通信内容
3. 冒充风险：黑客可以冒充他人身份参与通信。

HTTPS可以理解成HTTP+SSL/TLS，即HTTP下面加入SSL层，这是HTTPS的安全基础。用于安全的HTTP数据传输。

SSL（安全套接字）协议位于TCP/IP协议和各种应用层协议之间，为数据通讯安全提供支持。

1. 所有信息都是加密传播，第三方无法窃听。
2. 具有校验机制，一旦被篡改，通信双方会立即发现。
3. 配备身份证书，防止身份被冒充。

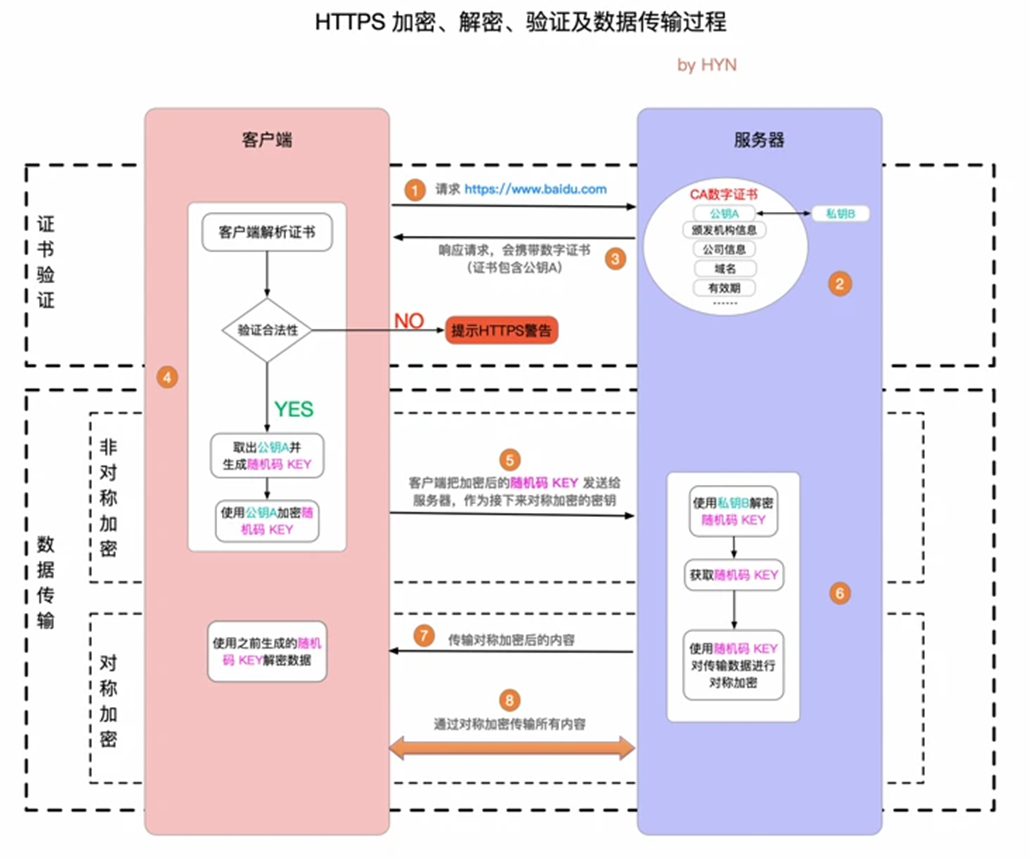


HTTPS的整个通信过程可分为两大阶段：**证书验证**和**数据传输阶段**，**数据传输阶段又分为非对称加密和对称加密两个阶段**。

1. **客户端请求HTTPS地址，连接到服务器的443端口号**（HTTP使用的默认端口是80）
2. **采用HTTPS协议服务器必须要有一套数字证书（CA）**，证书是需要申请的，并由专门的数字证书认证机构通过非常严格的审核之后颁发的。**颁发的同时会产生一个私钥和公钥。私钥由服务器自己保存，不可泄露。公钥附带在证书的信息中，可以公开**。证书本身也附带一个证书电子签名，这个签名用来验证证书的完整性和真实性，可以防止证书被篡改。
3. **服务器端响应客户端请求，把证书传输给客户端**，客户端收到的证书中包含有公钥、证书颁发机构信息、证书有效期等信息。
4. **客户端解析证书并对其进行验证**。如果证书不是可信机构颁布，或者证书中的域名和实际域名不一致，或者证书过期，就会向访问者显示一个警告。**如果证书没问题，客户端就取出服务器传来的公钥，用公钥加密自己的密钥**。
5. **客户端把加密后的密钥发送给服务器**。
6. **服务器收到客户端传来的加密密钥之后，用私钥将其解密**。

（经过上面步骤,解决了对称加密的密钥泄露问题）

1. 服务器用解密后得到的客户端密钥对数据进行对称加密并发给客户端，客户端使用相同的密钥解密数据
2. **双方使用对称加密传输所有数据。**



以上就是HTTPS的原理。个人认为这很好的结合了对称加密和非对称加密的优点。对称加密速度快、加密效率高、计算量小，但是在传输密钥的过程中是明文传输的，中间有可能被恶意窃取，这就造成了对称加密的不安全性。非对称加密安全，但是速度慢。于是HTTPS采用了以上方式。

总结HTTP和HTTPS的区别

* HTTP是不安全的，HTTPS是安全的
* HTTP工作于应用层，HTTPS的安全传输机制工作在传输层
* HTTP无法加密，HTTPS对传输的数据进行加密
* HTTP无需数字颁发证书，HTTPS需要CA机构wosign颁发的SSL证书

http常见状态码

**2xx：（请求成功）表示成功处理了请求的状态码**

200：(成功)服务器已成功处理了请求，

202：服务器已接收请求，但未处理

**3xx：（请求被重定向）表示要完成请求，需要进一步操作。**

301：（永久移动）请求的网页已永久移动到新位置，会自动将请求者转到新位置

304：（未修改）请求的网页自上次没有修改过。不会返回网页内容

4xx：（请求错误）这些状态码表示请求可能出错，妨碍了服务器的处理

400：服务器不理解请求的语法

403：（禁止）服务器拒绝请求

404：找不到请求的网页

5xx:（服务器错误）这些状态码表示服务器在尝试处理请求时发生内部错误。这些错误可能是服务器本身的错误

500：服务器内部错误

505：（HTTP版本不受支持）服务器不支持请求中所用的HTTP协议版本

**Http请求头包含哪些内容**

Host 请求的地址域名和端口，不包括协议

Connection //连接类型

User-Agent //浏览器的用户代理信息

Accept // 浏览器支持的请求类型

Accept-Encoding //浏览器能处理的压缩代码

Accept-Language //浏览器当前设置语言

**XSS是什么？**

XSS就是黑客想尽一切办法把恶意代码注入到网页中。

1. 存储型XSS

场景：见于带有用户保存数据的网站功能，如论坛发帖、商品评论、用户私信。

攻击步骤

1. 攻击者将恶意代码提交到目标网站的数据库
2. 用户打开目标网站时，服务器获取恶意代码并把它拼接到html中
3. 用户浏览器获得返回的html，解析执行，混在起中的恶意代码也会被执行
4. 恶意代码窃取用户数据，并把它发送到指定攻击者的网站，或者冒充用户行为，调用目标网站的接口，执行恶意操作
5. 反射型XSS

场景：通过URL传递参数的功能，如网站搜索、跳转等。

1. 攻击者构造特殊的URL，其中包含恶意代码
2. 用户跳转URL，服务器把恶意代码从URL中提取出来，拼接在html中返回给浏览器。
3. 浏览器收到响应后，解析html，其中的恶意代码也会被执行。
4. 恶意代码窃取用户数据，并把它发送到指定攻击者的网站，或者冒充用户行为，调用目标网站的接口，执行恶意操作。
5. DOM 型XSS

DOM型XSS攻击中，取出和执行恶意代码由浏览器端完成，属于前端JavaScript自身的安全漏洞，而其它两种XSS都属于服务端的安全漏洞。

场景：通过URL传递参数的功能，如网站搜素、跳转等。

1. 攻击者构造特殊URL，其中包含恶意代码。
2. 用户打开带有恶意代码的URL
3. 用户获取响应之后，前端JavaScript取出URL中的恶意代码并执行。
4. 恶意代码窃取用户数据并发送到攻击者网站，或者冒充用户行为，调用目标网站接口执行攻击者指定的操作。

**XSS避免方式：**

1. url参数使用encodeURIComponent方法转义
2. 尽量不使用innerHTML插入HTML内容
3. 使用特殊符号、标签转义符

**CSRF:跨站请求伪造**

攻击者诱导受害者进入第三方网站，在第三方网站中，向被攻击网站发起跨站请求。利用受害者在被攻击网站已经获取的注册凭证，绕过后台的用户验证，达到冒充用户对被攻击的网站执行某恶意操作。

攻击类型：

* GET：如在页面的某个img中发起一个get请求
* POST：通过自动提交表单到恶意网站
* 链接型：需要诱导用户点击链接

预防方案

CSRF通常从第三方网站发起，被攻击的网站无法防止攻击发生，只能通过增强自己网站针对CSRF的防护能力来提升安全性。

CSRF避免方式：

1. 添加验证码
2. 使用token

* 服务器给用户生成一个token，加密后传递给用户
* 用户在提交申请时，需要携带这个token
* 服务端验证token是否正确

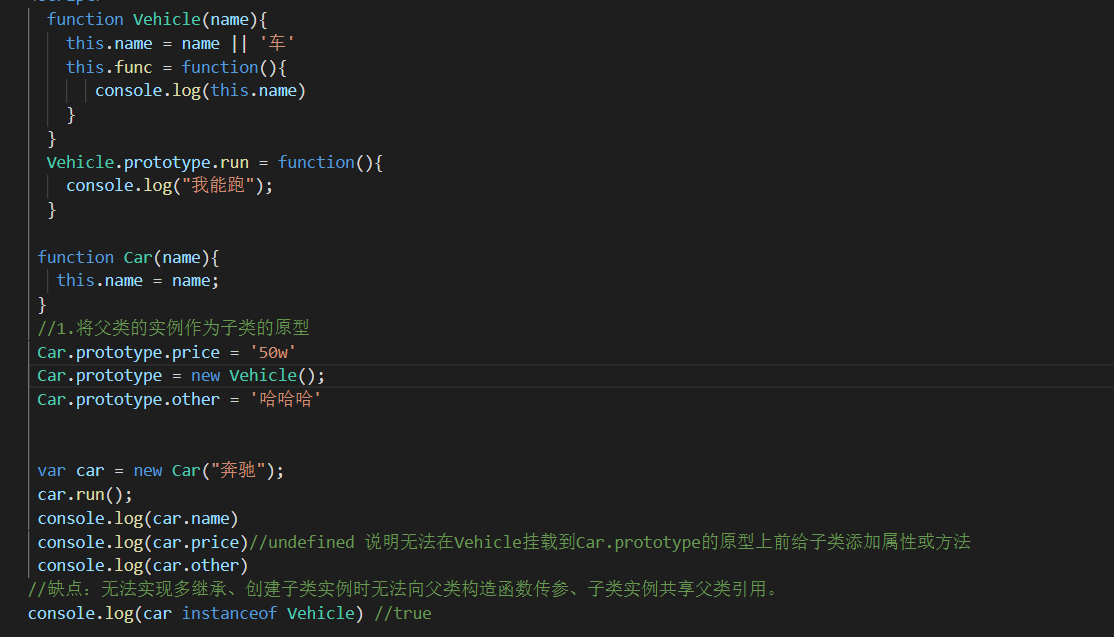
**JavaScript采用的是单线程模式**，也就是说一次只能做一件事。虽然H5中的Web Worker为JavaScript创建了多线程环境，但是子线程完全受主线程控制，而且不能操作DOM元素。这没有改变JavaScript的单线程机制，打个比方，如果JavaScript是多线程的，一个线程要删除一DOM元素，而另一线程要往这个DOM元素添加内容。那引擎该听谁的呢？JavaScript只能单线程，所以 JavaScript的所有任务都需要排队完成。JavaScript的任务又分为同步任何和异步任务。同步任务直接放在主线程中排队依次执行，异步任务交给浏览器处理，浏览器先在事件表注册异步任务，处理完成后再把它加入异步任务队列中。当主线程中的同步任务执行完，先在宏任务队列里取出一个宏任务放入调用栈中让主线程执行调用栈的任务，当执行过程中遇到微任务时，把微任务放入微任务队列里，这个宏任务执行完毕之后立即执行当前微任务队列中的所有微任务，如在执行的过程中又加入了微任务，依然执行微任务队列里微任务直到队列清空。常见的宏任务有setTimeout、script、setInterval，微任务有Promise.then(),catch finally,process.nextTick。那么**为什么要区分宏任务和微任务**呢？个人理解，区分这两种任务是为了插队。因为**Event Loop中微任务队列里所有任务执行完毕才能进行下一个Event Loop去执行一个宏任务**。当微任务在执行过程中又碰到一个微任务，这依然会在本次的Event Loop中执行，所以宏任务等。

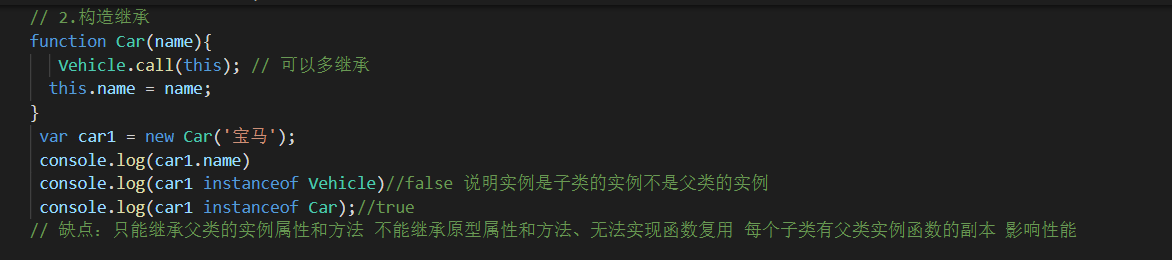
**JavaScript执行过程分为解析阶段和运行阶段**。解析阶段分为词法分析和语法分析。词法分析就是把代码中的字符串分割成token（词法单元）；语法分析即是将token流转换成一棵抽象语法树。在JS代码正式运行之前还会进行预解析的工作，比如进行变量提升、函数提升。这使得我们可以在变量赋值前使用变量，在函数声明之上调用函数。

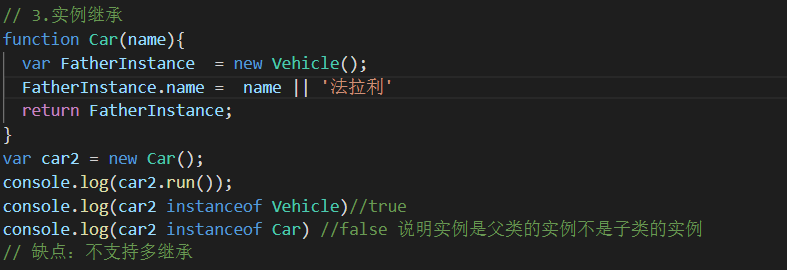
JavaScript也是一门基于面向对象思想创造的语言，它也有很多内置函数，比如用于解决异步编程回调地狱问题的Promise、或者Number、Boolean、Error、Date等等。如今于前端开发的http库——axios 的原理也是基于Promise实现的，还有Vue中会使用到各种实例化对象。但它并不是一门真正的面向对象语言，OOP的三大特性是：封装、继承和多态。前端的封装和后端的封装目的都是为了减少重复的劳动，把功能相似的封装起来，后续用到时直接引入调用。JAVA的继承意味着拷贝超类的所有公有的属性和方法，而JavaScript的继承是原型链上的关系，一个对象不会复制到其它对象，它们会被关联起来。prototype是Javascript中继承关系的联结点，Prototype和proto的区别是：Prototype是显式原型，只有构造函数拥有prototype。Prototype指向该对象的原型。当我们用.(点)语法或者getProperty访问子类实例的一个属性时，实际上，会先在该子类实例本身找，如果找不到，就通过\_\_proto\_\_这个纽带往prototype(原型对象)上就近原则查找，原型链的终点是Object.prototype == null，所有普通的prototype链都会指向内置的Object.prototype。调用Object.create()即凭空创建一个新对象并把它的proptotype和指定对象关联。Java的多态分为重载和重写两种，Java中的重载就是说，同名方法，不同的参数列表。JavaScript中没有重载的概念，同名方法会按赋值的顺序覆盖，用的是覆盖后的方法。Java的重写是子类重写父类的同名同参数列表的方法，子类实例化的对象调用同名方法时只用自己的方法，JavaScript的重写也许是改变原型链上原型对象的方法。

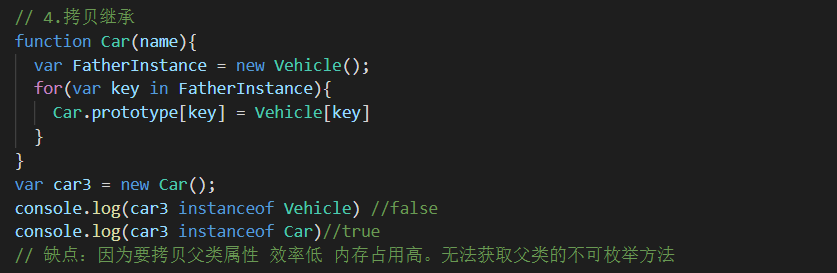
**JavaScript的基本数据类型**有：undefined、number、null、string、boolean、bigInt、Symbol。基本数据类型变量放在栈内存里，而且栈内存的大小是固定的，如果存储的大小大于栈内存大小就会溢栈。因为栈有先入后出的特点，所以方便查找基本数据类型变量。而引用类型数据放在堆内存中，堆内存是人为new分配的且人为释放的，运行效率低。栈内存的数据运行结束就被系统自动回收。堆内存的对象回收标准是否可达，在V8中对象先分配到新生代的From中，如果不可达直接释放，如果可达，就复制到To中，然后将To和From互换。当多次复制后依然没有回收，就放入老生代中，进行标记回收。之后将内存碎片进行整合放到一端。

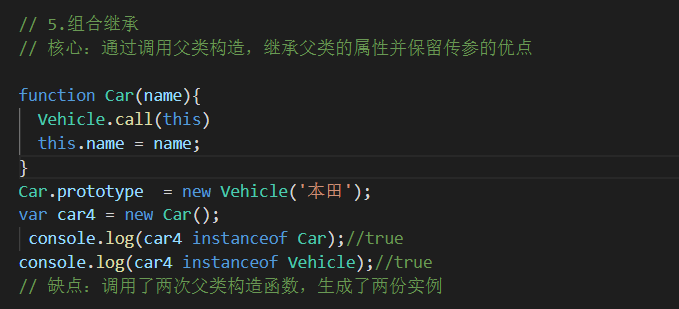
**JavaScript中实现继承**的方式又有哪些呢？

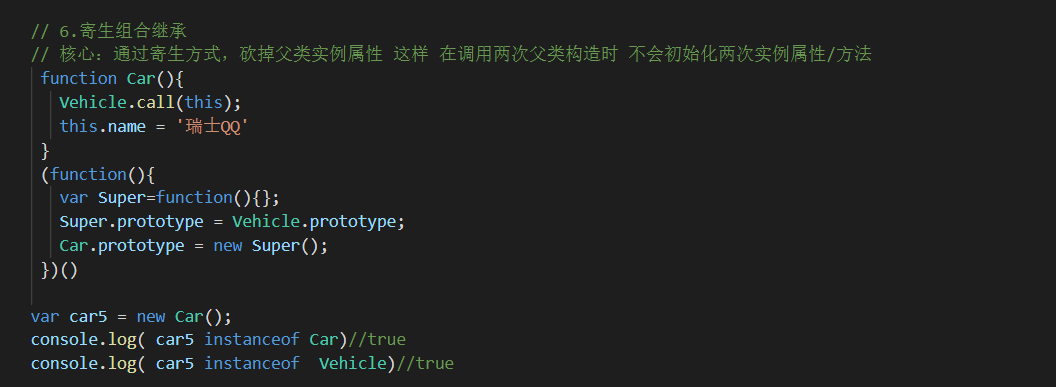












谈谈JavaScript学习过程中常被提及的闭包。说到闭包要了解**函数作用域**链。JavaScript中每一个方法里都有一个内置属性[[scope]]，它由JavaScript引擎存储管理，其实这个[[scope]]属性里也就是它的作用域链。JavaScript引擎会在页面加载后创建一个全局作用域，然后每碰到一个要执行的函数，就会为其创建对应的作用域，最终不同的块级作用域和嵌套在内部的函数作用域会形成一个作用域堆栈。当前生效的作用域在堆栈的最顶端，由上往下就是当前执行上下文所能访问的作用域链。**一个外部函数返回一个引用外部函数变量的内部函数，当外部函数执行完毕，作用域链销毁，但内部函数还引用着对外部函数的活动对象，这就形成了闭包**。闭包其实是一个特殊的函数，它可以访问函数内部的变量，并且让这些变量的值始终保持在内存中，不会在函数调用之后被垃圾回收机制清除。**闭包的优点**：1.可以从内部访问到外部变量，且这个变量常驻内存，可供之后使用。2.JavaScript在多人合作时，如果定义过多的全局变量，有可能会造成全局变量命名冲突，使用闭包可以避免变量污染全局。3.把变量存在独立的作用域，作为私有变量存在。**闭包的缺点**：1.有可能会获取到意外的值。2.因保持着对外部作用域活动变量的引用，造成内存泄露。3.闭包的层级决定了引用的外部变量在查找时经过的作用域链长度，层级太深对处理速度有影响。**闭包的应用场景一：**JavaScript设计模式中的模块模式，就是利用闭包封装私有状态和私有组织，对外只提供一个公共的API，其它的一切维持在私有闭包里。**闭包的应用场景二：**循环中创建闭包，防止取到意外的值。

**var let const三个关键字**

作用域：var声明的范围是函数作用域，let、const声明的是块级作用域

变量提升：var声明变量会提升到函数作用域顶部同时初始化，let、const变量提升到作用域顶部，在声明和初始化之间的区域叫做暂时性死区（禁止读写），所以在我们看来let、const不会变量提升

重复声明：var可以覆盖自己声明过的变量但不能重复声明覆盖let、const声明的变量。

var、let可以不赋初始值并且可以修改，const声明的时候必须初始化，而且不能修改地址。

Let、const相比var，在块级作用域比函数作用域更早终止的情况下有助于垃圾回收提升性能。

**数组的方法有哪些？**

1. Array.prototype.every

every()检测数组中每个元素是否都符合指定条件

1. Array.prototype.fill

fill()方法用于将一个固定值替换数组的元素

1. Array.prototype.some

Some方法用于判断是否有元素符合条件

1. forEach没有返回值 只对每个元素调用function
2. Array.prototype.find()

用来查找目标元素，找到就返回该元素，找不到就返回undefined

1. Array.prototype.findIndex()

查找目标元素，找到就返回元素的位置，找不到就返回-1

1. Array.prototype.includes()

用来判断一个数组是否包含一个指定的值，返回true or false

1. Array.prototype.indexOf()

方法可返回某个指定的字符串值在字符串中首次出现的位置

1. Array.prototype.join()

Join()方法用于把数组中所有元素连接成一个字符串

1. Array.prototype.reduce()方法用于接收一个函数callback作为累加器。数组中每个值从左到右开始合并，最终为一个值。
2. Array.prototype.reverse()方法用于颠倒数组中元素的顺序
3. Array.prototype.shift()用于把数组的第一个元素从其中删除，

返回第一个元素的值

1. Array.prototye.slice() 用于从已有数组中返回选定的元素，不改变原数组
2. Array.prototype.splice()用于删除和修改数组
3. Array.prototype.sort() 对数组的元素做原地排序，并返回这个数组。
4. Array.prototype.toLocaleString() 把数组转换为本地字符串
5. Array.prototype.toString() 把数组转换为字符串 并返回结果
6. Array.prototype.unshift() 向数组的开头添加一个或多个元素，并返回新的长度。

**如何判断this绑定？**判断this绑定需要知道函数的调用位置，分析调用栈，我们关心的调用位置就在当前正在执行的函数的前一个调用中。有如下四种绑定规则：一个是**默认绑定**。独立函数调用时采用默认绑定规则，严格模式下默认绑定undefined，非严格模式下默认绑定window。一个是**隐式绑定**。当函数引用有上下文对象时，隐式绑定会把this绑定到这个上下文对象中。有时候会发生丢失绑定对象的现象，丢失绑定对象就会采用默认绑定。一个是**显式绑定**。 当我们不想隐式绑定（在对象中包含函数引用），想让某个对象中强制调用某个函数，就可以采用显式绑定的方法。使用call(),apply()。还有一种叫**new绑定**。在Java中，使用new初始化一个类时会调用类的构造函数。而JavaScript中,new不会实例化一个类，只是用new操作符调用这一个函数，它不是传统的构造函数。当使用new调用函数时会发生如下：首先，会构造一个全新对象。接着，这个新对象会被原型连接到调用的函数。然后这个新对象被绑定到函数调用的this。最后，如果函数没有返回别的对象，new表达式中的函数调用会自动返回这个新对象。**This绑定的优先级**：4321。By the way,箭头函数中的并不采用如上四种绑定规则，因为箭头函数没有自己的this,它的this继承自外层函数调用的this。

箭头函数和普通函数的区别除了上述的this指向之外，还有以下几点：1.箭头函数没有argument属性。2.箭头函数不能作为构造器，因此不能通过new来调用，所以它没有new.target属性。3.不能使用call，apply,bind，通过显式绑定修改箭头函数的this指向。4.箭头函数不能作为generator函数。5.箭头函数的this指向箭头函数定义时所处的对象，而不是箭头函数使用时所在的对象。

什么是回调地狱？说起回调地狱，首先要知道什么是异步，举个例子，当我们去肯德基点餐，点餐结束拿到单据之后我们不需要傻愣着等待出餐，可以坐着玩会手机，等服务员叫号了再去取餐。简言之就是发起者不关心事件的状态，不需要等待接收者返回信息。在ES6之前，我们通过回调实现异步逻辑。回调函数是作为参数传递给另一个函数的函数，这个函数会在另一个函数执行后返回。**Callback回调的优点**：不会因为响应过慢阻塞后面的任务。**缺点**：1.缺乏顺序性,不是大脑的思维方式。2.嵌套太多很难处理错误。3.嵌套函数之间存在耦合性。我们常说的回调地狱就是一个异步事件嵌套着另一个异步事件，一个异步事件依赖于另一个异步事件，一旦嵌套层级多了，代码结构就容易变得不直观。代码执行的顺序不是按视觉的由上自下，而是基于逻辑的完成时机跳跃式触发。

ES6的Promise是异步编程的一种解决方案，比起回调函数，更合理更强大。有了Promise，就可以将异步操作用同步操作的方式表达出来，避免层层嵌套的回调函数。

Promise就是一个异步容器。 它有以下特点：

1. **对象的状态不受外界影响。**Prmise只有三种状态：pending、fufilled、rejected。只有异步操作的结果可以决定当前是哪一种状态。任何其他操作都无法改变这个状态。
2. **一旦状态改变就不会再变。**Promise对象状态改变只有两种可能：从pending到fullilled和从pending到rejected。
3. **Promise里是同步执行的，无法取消，then里面是异步的，属于微任务。**

**Promise有关的api**

1. **Promise.all()**

1.接收一个数组/具有Iterator接口的作为参数，数组里是promise实例，如不是，就先调用Promise.resolve方法，将参数转为Promise实例再进一步处理

2.成功：**参数的promise状态都变成fulfilled，Promise.all()返回的结果才会变成fulfilled**，此时参数数组里的每个promise的返回值组成一个数组，传递给Promise.all()的结果的回调函数。

3.失败：**其中一个promise变为rejected，就会调用Promise.all()方法后面的回调函数**。(如果作为参数的promise实例，自己定义了catch方法,那么它一旦被rejected，并不会触发Promise.all()的catch方法)

1. **Promise.race()**
2. 同样是将多个Promise实例包装成一个新的Promise实例
3. 如果参数里的不是promise实例，就先调用Promise.resolve()方法，将参数转为Promise实例，再进一步处理。
4. **只要有一个实例率先改变状态，结果的状态就会跟着改变，那个率先改变的promise实例的返回值，就传递给p的回调函数**。
5. **Promise.allSettled()**
6. Promise.all()可以确定所有请求都成功了，但是只要有一个请求失败，它就会报错，而不管另外的请求是否结束。
7. **Promise.allSettled()用来确定一组异步操作是否都结束了**。所以它的名字叫Settled，包含了fulfilled和rejected两种情况。
8. 它接受一个数组作为参数，数组的每个成员都是一个Promise对象，并返回一个新的Promise对象。
9. **只有等到参数数组所有的Promise对象都发生状态变更，返回的Promise对象才会发生状态变更**。

**Async函数**

**Async函数的实现原理，就是将Generator函数和自动执行器包装在一个函数里**

一般情况下async/await在错误处理方面，主要使用try/catch，但是如果有多个异步操作，就需要写多个try catch，这种方式间接性很差。

Async/await 本质就是promise的语法糖，既然是promise就可以使用then函数

 // 抽离成公共方法

   function errorHandle(promise){

    return promise.then( data => {

      return [null,data];

    }).catch(

       err => {

         return [err,null]

       }

    )

一、区别

1.定义

* Promise是对象，用于表示异步操作的最终完成或失败，及其结果值
* Async function 是声明语句，用来定义一个返回AsyncFunction对象的异步函数，它会通过一个隐式的Promise返回其结果
* Await是表达式，用于暂停当前异步函数的执行，等待Promise处理完成，只在异步函数内有效。

错误的捕获

2.错误的捕获

* 使用async/await语法能让开发者在异步代码中也按照类似同步代码一样的方式使用try-catch语句
* Promise的错误无法在外部被捕捉到，只能在内部进行预判处理

3.中断

* Promise是一个状态机，本身无法完全中止
* Async function 使用语义化的await中断程序

二、关系

Async/await的目的是简化使用多个Promise时的同步行为，并对一组Promises执行某些操作。正如Promises类似于结构化回调，async/await更像结合了generators和promises。Async/await是generator的语法糖，它产生一个promise。

* 1. 使用场景

一般在异步处理时使用，而通常的，由于它们之间紧密的关系，都是配合使用。

1. Promise提供的工具函数对应的场景
2. Async/await避免了繁杂的Promise链式调用，更加语义化，推荐使用async/await。
3. Await将异步代码改成同步，如果这些异步代码之间没有依赖，await使性能降低。

**重绘和回流**是什么？重绘是页面布局没有改变，元素的几何尺寸和位置不变，比如修改颜色。回流意味着元素的几何尺寸变了，我们需要重新验证并计算渲染树。

**垃圾回收机制**：

JavaScript有自动垃圾回收机制。也就是说，执行环境会负责管理内存的分配和回收。JavaScript的自动垃圾回收机制原理就是找到不再用到的变量，释放它们所占用的内存。所以，垃圾收集器会按固定的时间间隔，周期性的执行这一操作。

垃圾收集器需要找到这些不再使用的变量，对这些不再使用的变量打上标记，以便后面回收它们的内存。JavaScript的垃圾回收机制通常有两种策略：一是标记清除法，一是引用计数法。

最常用的回收策略是标记清除法。当一个变量进入执行环境，就会这个变量打上“进入环境”的标记；当一个变量离开执行环境，就对这个变量打上“离开环境的标记”。从理论上来说，进入环境的变量永远不会被回收内存，因为执行流进入环境时，这些变量可能会被用到。**可以翻转某个特殊的位来记录变量何时进入环境，或者使用一个“进入环境”列表和一个“离开环境”列表跟踪变量的变化**。**垃圾收集器会对存储在内存中的所有变量打上标记。然后，它会去掉环境中的变量和被环境中的变量引用的变量的标记**。再次之后再被加上标记的变量将被视为准备删除的变量。原因是环境中的变量已经无法再访问这些变量了。最后垃圾收集器完成内存清除工作。

还有一种不常用的回收策略是引用计数法。引用计数法顾名思义就是记录变量的引用次数。当引用这个变量，这个变量的引用次数+1，当原来引用这个变量的变量再取了别的变量，这个变量的引用次数-1。当引用次数为0时，说明没办法再访问这个值。因此就可以将其占用的内存空间回收回来。这样做会带来一个很大的问题，就是循环引用的变量无法被回收。比如A引用B，B又引用A，A和B的引用次数永远不可能为0。

**Tree-shaking的原理**

当前端项目达到一定规模后，我们一般会采用按模块方式组织代码，这样可以方便代码的组织和维护。但会存在一个问题，比如我们有一个utils工具类，在另一个模块中导入它。这会在打包的时候将utils中不必要的代码也打包，从而使得打包体积变大，这时候就需要Tree-shaking技术了。

**Tree-shaking是一种通过清除多余代码方式来优化项目打包体积**。

**原理：利用ES6模块的特点**

* 只能作为模块顶层的语句出现。
* import的模块名只能是字符串常量，不能动态引入模块。
* import binding是immutable的。引入的模块不能再进行修改**。**

虽然tree-shaking的概念在1990提出，但直到ES6的ES6-style模块出现后才真正被利用起来。这是因为tree-shaking只能在静态modules下工作。ES6中使用tree-shaking是非常容易的。而且tree-shaking不仅支持import/export级别，也支持statement级别。

在ES6以前，我们使用CommonJS引入模块：require()的引入是运行时载入，意味着我们可以基于条件来导入所需要的代码。CommonJS的动态特性模块意味着tree-shaking不适用，因为它无法确定哪些模块实际运行之前是需要的或者不需要的。在ES6中，进入了完全静态的导入语法：import。ES6的import语法完美可以使用tree-shaking。因为可以在代码不运行的情况下就能分析出不需要的代码。

从webpack2开始支持实现了Tree-shaking的特性，webpack2正式版本内置支持ES5模块和未引用模块检测能力，通过package.json的sideEffects属性作为标记，向compiler提供提示，表明项目中的哪些文件是“pure”的，由此可以安全地删除文件中未使用的部分。

如果使用的是webpack4，只需要将mode设置为production即可开启tree-shaking。如果是使用webpack2，tree-shaking不起作用。因为babel会将代码编译成CommonJs模块，而tree-shaking不支持CommonJS。所以需要配置不转义。

**Side effects(副作用)**

**Side effects指那些当import的时候会执行一些动作，但是不一定会有任何export。比如ployfill,ployfills不对外暴露方法给主程序使用。**

Tree-shaking不能自动的识别那些代码属于side effects，因此手动指定这些代码非常重要。如果不指定可能会出现一些意向不到的问题。

在webpack中，是通过package.json的sideEffects属性。如果所有代码都不包含副作用，我们就可以简单的将该属性标记为false，来告知webpack，它可以安全地删除未用到的export导出。

总结：

* tree-shaking不支持动态导入，只支持纯静态导入import/export。
* webpack中可以在项目package.json文件中，添加一个”sideEffects”属性，手动指定有副作用的脚本。

**前端模块化，CMD、AMD、CommonJS**

**CommonJS**是服务器端模块的规范，分为：模块定义、模块标识、模块引用。

模块定义：每一个模块中，module对象代表该模块自身。export属性：module对象的一个属性，它向外提供接口。输出模块变量的最好方法是使用module.exports对象。一个单独的文件就是一个模块。每一个模块都是一个单独的作用域，也就是说，在该模块内部定义的变量，无法被其他模块读取，除非定义为global对象的属性。

模块标识：传递给require方法的参数，必须是符合小驼峰命名的字符串，或者以.、..、开头的相对路径，或绝对路径。

模块引用：加载模块使用require(同步加载)，该方法读取一个文件并执行，返回文件内部的module.exports对象。

优势：

* CommonJS模块规范很好地解决变量污染问题，每个模块局具有独立空间，互不干扰，命名空间相比之下就不太好
* CommonJS规范定义模块十分简单，接口十分简洁
* CommonJS模块规范支持引入和导出功能，这样可以顺畅地连接各个模块，实现彼此间的依赖关系
* CommonJS规范的提出，是为了弥补JavaScript没有标准的缺陷，具备开发大型应用的基础能力。而不是停留在开发浏览器端小脚本程序阶段。

缺陷：

**AMD**

AMD推崇依赖前置，在定义模块的时候就要声明其依赖的模块

优点：用户体验好，因为没有延迟，依赖模块提前执行了

**CMD规范**

CMD推崇依赖就近，只有在用到某个模块的时候才会去require

优点：

性能好，因为只有用户需要的时候才执行。

多路复用代替原来的序列和阻塞机制，所有的请求都通过一个TCP连接并发完成

在HTTP2中

* 同域名下所有通信都在单个连接下完成，同个域名只需要占用一个TCP连接，使用一个连接并行发送多个请求和响应。
* 单个连接可以承载任意数量的双向数据流，单个连接上可以并行交错的请求和响应，之间互不干扰
* 数据流以消息的形式发送，而消息又由一个或多个帧组成，多一个帧之间可以乱序发送，因为根据帧首部的流标识可以重新组装

Keep-Alive解决的核心问题是：一定时间内同一域名多次请求数据，只建立一次HTTP请求，其它请求可以复用每一次建立的连接通道，以达到提高请求效率的问题。

**JavaScript的假值有哪些？**

1. False
2. Null
3. Undefined
4. 0
5. ‘’
6. NaN

**XSS**是跨站脚本攻击、向目标网站插入恶意代码、大量用户访问网站时运行恶意脚本获取信息

【参考答案】

标准回答

XSS是跨站脚本攻击(Cross Site Scripting)，不写为CSS是为了避免和层叠样式表（Cascading Style Sheets）的缩写混淆，所以将跨站脚本攻击写为XSS。攻击者可以通过向Web页面里面插入script代码，当用户浏览这个页面时，就会运行被插入的script代码，达到攻击者的目的。XSS的危害一般是泄露用户的登录信息cookie，攻击者可以通过cookie绕过登录步骤直接进入站点。XSS的分类分为反射型和存储型。反射型就是临时通过url访问网站，网站服务端将恶意代码从url中取出，拼接在HTML中返回给浏览器，用户就会执行恶意代码。存储型就是将恶意代码以留言的形式保存在服务器数据库，任何访问网站的人都会受到攻击。预防XSS攻击的方案基本是对数据进行严格的输出编码，比如HTML元素的编码，JavaScript编码，css编码，url编码等等。

加分回答

XSS的危害：

• 获取cookie：网站中的登录一般都是用cookie作为某个用户的身份证明，这是服务器端返回的一串字符。如果cookie被攻击者拿到，那么就可以绕过密码登录。当空间、论坛如果可以被插入script代码，那么进入空间或者论坛的人的账号就可以轻易被攻击者获取。

• 恶意跳转：直接在页面中插入window.location.href进行跳转。

XSS的分类：

• 反射型XSS（非持久型XSS）：通过URL参数直接注入

• 存储型XSS（持久型XSS）：存储到数据库后读取时注入

XSS的预防：

• 浏览器的防御和“X-XSS-Protection”有关，默认值为1，即默认打开XSS防御，可以防御反射型的XSS，不过作用有限，只能防御注入到HTML的节点内容或属性的XSS，例如URL参数中包含script标签。不建议只依赖此防御手段。

• 防御HTML节点内容，通过转义<为&lt以及>为&gt来实现防御HTML节点内容。

• 预防HTML属性，通过转义"->&quto来实现防御，一般不转义空格，但是这要求属性必须带引号。

• 预防JavaScript代码，通过将数据进行JSON序列化。

• 防御富文本是比较复杂的工程，因为富文本可以包含HTML和script，这些难以预测与防御，建议是通过白名单的方式来过滤允许的HTML标签和标签的属性来进行防御，大概的实现方式是：

o 将HTML代码段转成树级结构的数据

o 遍历树的每一个节点，过滤节点的类型和属性，或进行特殊处理

o 处理完成后，将树级结构转化成HTML代码

• 开启浏览器XSS防御：Http Only cookie，禁止 JavaScript 读取某些敏感 Cookie，攻击者完成XSS注入后也无法窃取此 Cookie。

【延伸阅读】

举个简单的例子就是一个留言板，如果攻击者在留言板中写入一个被script标签插入的代码，那么这个时候问题就来了，浏览器解析到留言内容的时候会把script当作浏览器脚本，直接运行其中的内容而不是通过渲染进程将script标签和里面的内容渲染成内容。

**JS异步实现的方法**

概括： JS 异步编程进化史：callback -> promise -> generator/yield -> async/await。

所有异步任务都是在同步任务执行结束之后，从任务队列中依次取出执行。

回调函数是异步操作最基本的方法，比如AJAX回调，回调函数的优点是简单、容易理解和实现，缺点是不利于代码的阅读和维护，各个部分之间高度耦合，使得程序结构混乱、流程难以追踪（尤其是多个回调函数嵌套的情况），而且每个任务只能指定一个回调函数。此外它不能使用 try catch 捕获错误，不能直接 return

* Promise包装了一个异步调用并生成一个Promise实例，当异步调用返回的时候根据调用的结果分别调用实例化时传入的resolve 和 reject方法，then接收到对应的数据，做出相应的处理。Promise不仅能够捕获错误，而且也很好地解决了回调地狱的问题，缺点是无法取消 Promise，错误需要通过回调函数捕获。
* Generator 函数是 ES6 提供的一种异步编程解决方案，Generator 函数是一个状态机，封装了多个内部状态，可暂停函数, yield可暂停，next方法可启动，每次返回的是yield后的表达式结果。优点是异步语义清晰，缺点是手动迭代Generator 函数很麻烦，实现逻辑有点绕
* async/await是基于Promise实现的，async/await使得异步代码看起来像同步代码，所以优点是，使用方法清晰明了，缺点是await 将异步代码改造成了同步代码，如果多个异步代码没有依赖性却使用了 await 会导致性能上的降低，代码没有依赖性的话，完全可以使用 Promise.all 的方式。

async/await函数对 Generator 函数的改进，体现在以下三点：

· 内置执行器。Generator 函数的执行必须靠执行器，而 async 函数自带执行器。也就是说，async 函数的执行，与普通函数一模一样，只要一行。

· 更广的适用性。yield 命令后面只能是 Thunk 函数或 Promise 对象，而 async 函数的 await 命令后面，可以跟 Promise 对象和原始类型的值（数值、字符串和布尔值，但这时等同于同步操作）。

· 更好的语义。async 和 await，比起星号和 yield，语义更清楚了。async 表示函数里有异步操作，await 表示紧跟在后面的表达式需要等待结果。

目前使用很广泛的就是promise和async/await

**Token可以存储在cookie中吗?**

能、不设置cookie有效期、重新登录重写cookie覆盖原来的cookie

【参考答案】

标准回答

能。

token一般是用来判断用户是否登录的，它内部包含的信息有：uid(用户唯一的身份标识)、time(当前时间的时间戳)、sign（签名，token 的前几位以哈希算法压缩成的一定长度的十六进制字符串）

token可以存放在Cookie中，token 是否过期，应该由后端来判断，不该前端来判断，所以token存储在cookie中只要不设置cookie的过期时间就ok了，如果 token 失效，就让后端在接口中返回固定的状态表示token 失效，需要重新登录，再重新登录的时候，重新设置 cookie 中的 token 就行。

加分回答

token认证流程

1. 客户端使用用户名跟密码请求登录

2. 服务端收到请求，去验证用户名与密码

3. 验证成功后，服务端签发一个 token ，并把它发送给客户端

4. 客户端接收 token 以后会把它存储起来，比如放在 cookie 里或者 localStorage 里

5. 客户端每次发送请求时都需要带着服务端签发的 token（把 token 放到 HTTP 的 Header 里）

6. 服务端收到请求后，需要验证请求里带有的 token ，如验证成功则返回对应的数据